# Message Passing Interface: Basic Course

Jerry Eriksson, Mikael Rännar and Pedro Ojeda

HPC2N,
UmeåUniversity,


901 87, Sweden.

April 23, 2015

# Table of contents

# Application of Parallel algorithms

## Molecular Dynamics

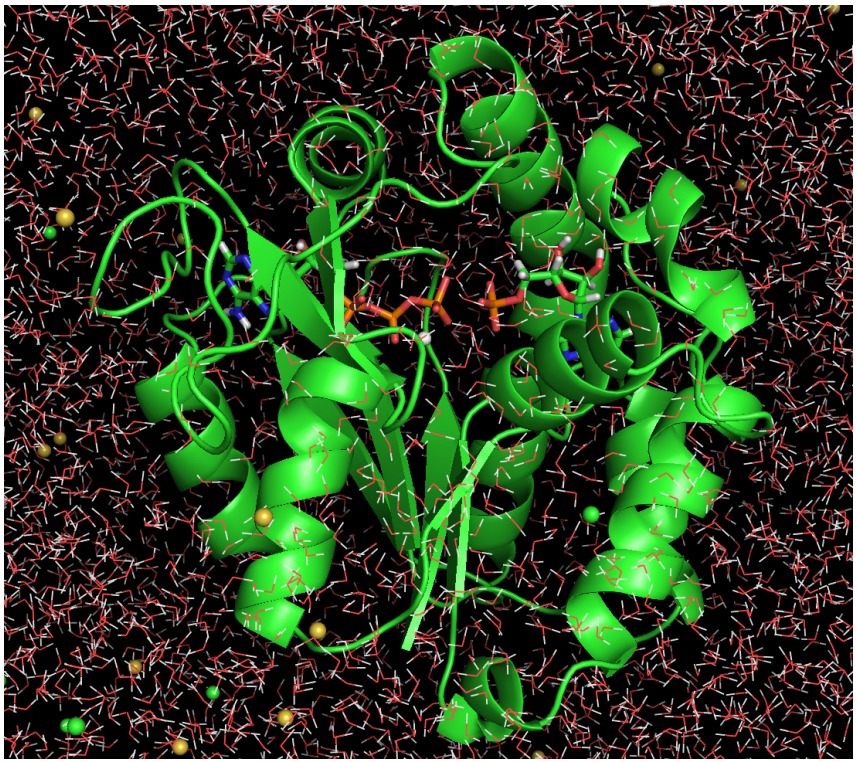

Figure : AdK enzyme in water.

## Simulations of Galaxies properties

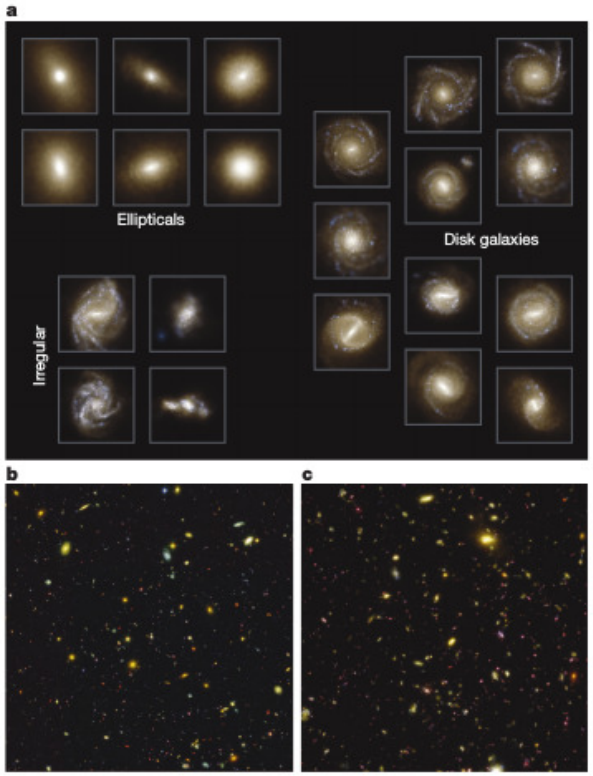


Figure : Galaxies [Nat., **509**, 177

# Working with arrays

$$\mathbf{F} = -\nabla U \qquad \text{Newton's Law} \tag{1}$$

solution of this equation requires the knowledge of an array of particles' positions and velocities

$$\mathbf{X} = (\ x_1^1, x_2^1, x_3^1\ ,\ x_1^2, x_2^2, x_3^2\ \ldots\ x_1^N, x_2^N, x_3^N\ ) \tag{2}$$

$$\mathbf{V} = (\ v_1^1, v_2^1, v_3^1\ ,\ v_1^2, v_2^2, v_3^2\ \ldots\ v_1^N, v_2^N, v_3^N\ ) \tag{3}$$

# Working with arrays

$$\mathbf{F} = -\nabla U \qquad \text{Newton's Law} \tag{1}$$

solution of this equation requires the knowledge of an array of particles' positions and velocities

$$\mathbf{X} = (\boxed{x_1^1, x_2^1, x_3^1}, \boxed{x_1^2, x_2^2, x_3^2} \ldots \boxed{x_1^N, x_2^N, x_3^N}) \tag{2}$$

$$\mathbf{V} = (\boxed{v_1^1, v_2^1, v_3^1}, \boxed{v_1^2, v_2^2, v_3^2} \ldots \boxed{v_1^N, v_2^N, v_3^N}) \tag{3}$$

# Distributed Memory and Message-Passing

- Each **process** has a separate address space
- Processes communicate by explicitly sending and receiving **messages**

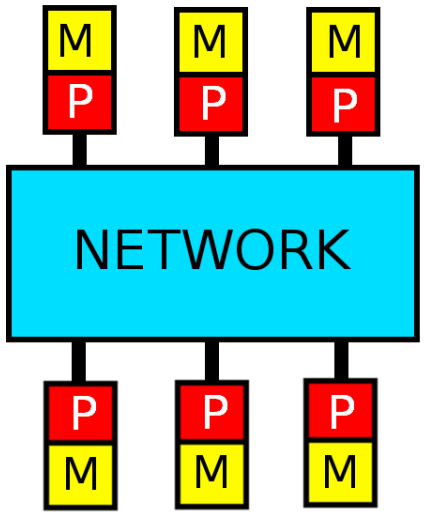


Figure : Distributed memory.

Figure : Abisko at HPC2N.

# Running jobs on Abisko

- Load Modules
- Compiling and linking
- Testing MPI programs
- Job submission

# Modules

```
# OpenMPI for the PathScale compiler
module load openmpi/psc

# OpenMPI for the GCC compiler
module load openmpi/gcc

# OpenMPI for the Portland compiler
module load openmpi/pgi

# OpenMPI for the Intel compiler
module load openmpi/intel
```

# Compiling and linking

- MPI provides a **compiler wrapper** named `mpicc/mpif90`
- Compile and link using the wrapper

**Example:**

```
# Compile mysrc.c
mpicc -std=c99 -c mysrc.c
# Compile main.c
mpicc -std=c99 -c main.c

# Link run.x
mpicc/mpif90 -o run.x main.o mysrc.o
```

# Testing MPI programs

- You can test your MPI programs on the login node
- Use the `mpirun` command to launch
  - Will give you some warnings but don't be alarmed
  - Only for short-running jobs
  - Cannot be used to test performance

**Example:**

```
# Launch run.x with four processes on login node
mpirun -np 4 ./run.x
```

# Job submission

**Template for a job script (`script.sbatch`):**

```
#!/bin/bash
#SBATCH -A SNIC2015-7-15
#SBATCH --reservation SNIC2015-7-15
#SBATCH -n 16
#SBATCH --time=00:30:00
#SBATCH --error=job-%J.err
#SBATCH --output=job-%J.out

echo "Starting at `date`"
srun ./run.x
echo "Stopping at `date`"
```

**Job submission:**

```
sbatch script.sbatch
```

# Querying and cancelling jobs

```
# Get the status of all your jobs
squeue -u <user>

# Get the predicted start of your queued jobs
squeue -u <user> --start

# Cancel a job
scancel <jobid>
```

# MPI: Message Passing Interface

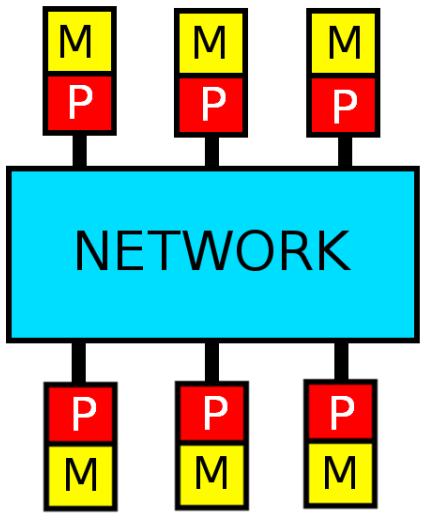


Figure : Distributed memory.

- MPI is the mostly used message passing-standard.
- There are many implementations, MPICH, MVAPICH, OpenMPI, etc.

# MPI: Message Passing Interface

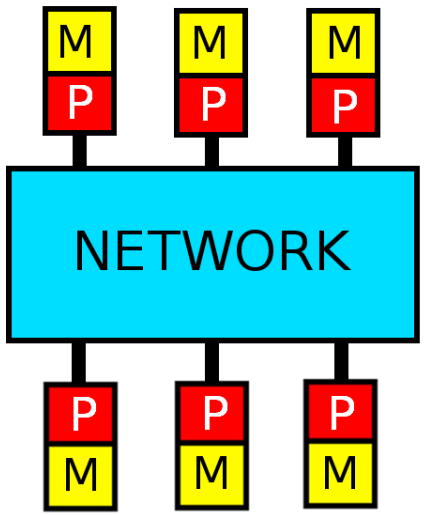


Figure : Distributed memory.

- MPI is the mostly used message passing-standard.
- There are many implementations, MPICH, MVAPICH, OpenMPI, etc.
- Supports Fortran, C, C++, etc.

# MPI: Message Passing Interface

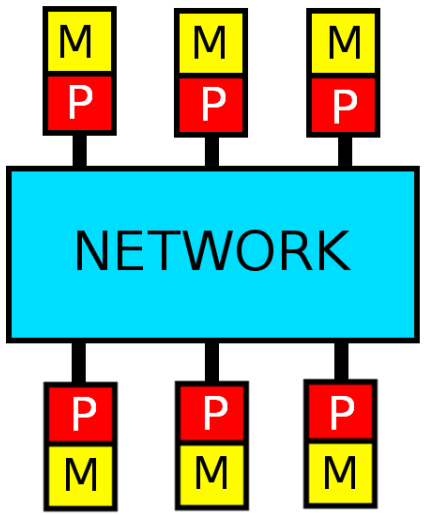


Figure : Distributed memory.

- MPI is the mostly used message passing-standard.
- There are many implementations, MPICH, MVAPICH, OpenMPI, etc.
- Supports Fortran, C, C++, etc.

# MPI Resources

- **Official MPI website**: `http://www.mpi-forum.org`
- **Specification (MPI version 2.2)**: `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`

# Scope of MPI

- Point-to-point message passing
- Collective communication
- One-sided communication
- Parallel I/O

# The Big Six

**The 6 core functions in MPI:**

- `MPI_Init`
  Initializes the MPI runtime system.
- `MPI_Finalize`
  Cleans up the MPI runtime system.
- `MPI_Comm_size`
  Returns the number of processes.
- `MPI_Comm_rank`
  Returns the rank (identifier) of the caller.
- `MPI_Send`
  Send a message.
- `MPI_Recv`
  Receive a message.

# Runtime system management

- Every MPI program must begin by calling `MPI_Init` and end by calling `MPI_Finalize`.
- `MPI_Init` takes the command line as parameters in order to process command line arguments that are understood by and intended for the MPI runtime system.
- `MPI_Finalize` takes no parameters and shuts down the runtime system.

# MPI C program template

```
// Include MPI-related declarations
#include <mpi.h>

int main( int argc, char *argv[] )
{
  // Initialize the MPI runtime system
  MPI_Init( &argc, &argv );

  // ..code that uses MPI..

  // Finalize the MPI runtime system
  MPI_Finalize( );
  return 0;
}
```

# MPI fortran program template

```
program main
 use MPI
 integer :: ierr,rank,size

 call MPI_INIT( ierr )
 call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr)
 call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr)
 ...
 call MPI_FINALIZE( ierr )

end
```

# Number of processes and process ranks

- Processes are distinguished by their **ranks**.
- The rank of a process is a number between 0 and $p-1$, where $p$ is the number of MPI processes.
- The number of processes and the rank of the caller can be obtained through the functions `MPI_Comm_size` and `MPI_Comm_rank`, respectively.

**Example:**

```
int np;
MPI_Comm_size( MPI_COMM_WORLD, &np );

int me;
MPI_Comm_rank( MPI_COMM_WORLD, &me );
```

# MPI_COMM_WORLD

- **Communicators** is the MPI term for communication contexts.
- The constant `MPI_COMM_WORLD` refers to a pre-defined communicator containing all MPI processes.
- For now, we always use the world communicator.

# Point-to-point communication

Only a sender and a receiver are involved in the communication.

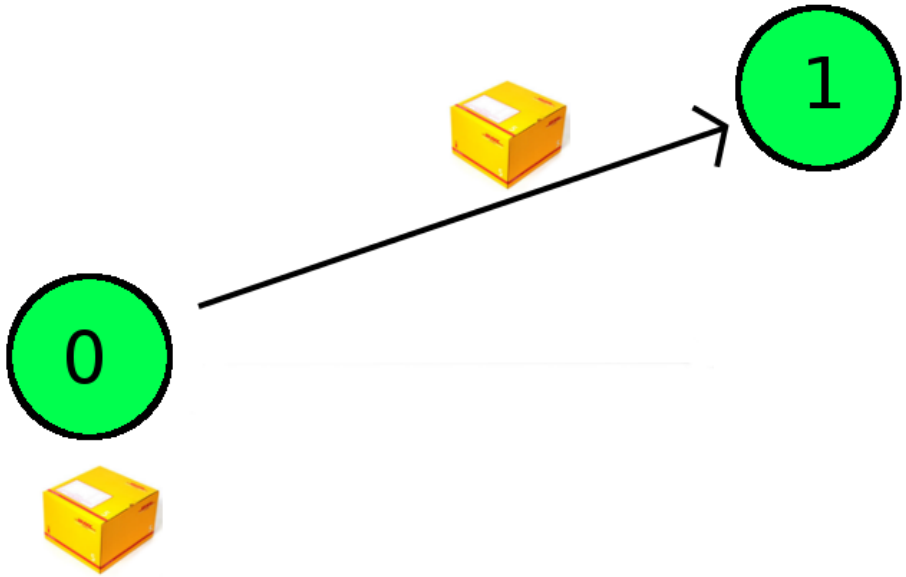


Figure : Point-to-point communication.

# Sending a message

- **Point-to-point messages can be sent** via the function `MPI_Send`
- An input buffer for the message data
- The number of elements in the message
- The element datatype
- The destination rank
- An identifying tag
- A communicator

**Example:**

```
char message[ 30 ] = "Hello MPI!";
MPI_Send( message, 30, MPI_CHAR,
          0,23, MPI_COMM_WORLD );
```

- Sends 30 character elements to rank 0 with tag 23 in the world communicator.

# Receiving a message

- **Point-to-point messages can be received** via the function `MPI_Recv`
- It takes the following parameters:
  - An output buffer for the message data
  - The maximum number of elements to receive
  - The element datatype
  - The source rank
  - An identifying tag
  - A communicator
  - An output status object

**Example:**

```
char message[ 30 ];
MPI_Recv( message, 30, MPI_CHAR,
          14, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE );
```

# MPI "Hello World" C Example

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main( int argc, char *argv[] )
5  {
6    int np, me;
7    MPI_Init( &argc, &argv );
8    MPI_Comm_size( MPI_COMM_WORLD, &np );
9    MPI_Comm_rank( MPI_COMM_WORLD, &me );
10   if( me == 1 ) {
11     char message[ 30 ] = "Hello␣MPI!";
12     MPI_Send( message, 30, MPI_CHAR,
13               0, 0, MPI_COMM_WORLD );
14   } else if( me == 0 ) {
15     char message[ 30 ];
16     MPI_Recv( message, 30, MPI_CHAR,
17               1, 0, MPI_COMM_WORLD,
18               MPI_STATUS_IGNORE );
19     printf( "Rank=0␣received␣\"%s\"\n", message );
20   }
21   MPI_Finalize( );
```

# Blocking/Non-blocking Communication

- `MPI_SEND` and `MPI_RECV` block execution until message is received.
- `MPI_ISEND` and `MPI_IRECV` provide a non-blocking execution.
- using non-blocking communications requires sync with `MPI_WAIT`

# Blocking Communication

```
if(rank == 0) then
  do i=1,nelem
      a(i)=i
      b(i)=2*i
  enddo
  write(6,10) 'rank 0 a',(a(i),i=1,10)
  write(6,10) 'rank 0 b',(b(i),i=1,10)
  call MPI_SEND(a,nelem,MPI_INT,1,1,MPI_COMM_WORLD,ierr)
  call MPI_SEND(b,nelem,MPI_INT,1,2,MPI_COMM_WORLD,ierr)
elseif(rank ==1) then
  call MPI_RECV(b,nelem,MPI_INT,0,2,MPI_COMM_WORLD,status,ierr)
  write(6,10) 'rank 1 b',(b(i),i=1,10)
  call MPI_RECV(a,nelem,MPI_INT,0,1,MPI_COMM_WORLD,status,ierr)
  write(6,10) 'rank 1 a',(a(i),i=1,10)
else
  print *, 'no other task'
endif
```

# Non-Blocking Communication

```
if(rank == 0) then
  do i=1,ne
      a(i)=i
      b(i)=2*i
  enddo
  write(6,10) 'rank 0 a',(a(i),i=1,10)
  write(6,10) 'rank 0 b',(b(i),i=1,10)
  call MPI_ISEND(a,ne,MPI_INT,1,1,MPI_COMM_WORLD,ierr)
  call MPI_ISEND(b,ne,MPI_INT,1,2,MPI_COMM_WORLD,ierr)
elseif(rank ==1) then
  call MPI_IRECV(b,ne,MPI_INT,0,2,MPI_COMM_WORLD,req,ierr)
  call MPI_WAIT(req, status, ierror)
  write(6,10) 'rank 1 b',(b(i),i=1,10)
  call MPI_IRECV(a,ne,MPI_INT,0,1,MPI_COMM_WORLD,req,ierr)
  call MPI_WAIT(req, status, ierror)
  write(6,10) 'rank 1 a',(a(i),i=1,10)
else
```

# Blocking Ring Communication

```
if(rank .ne. 0) then
  call MPI_RECV(trans,1,MPI_INT,rank-1,0,MPI_COMM_WORLD, &
     status,ierr)
  write(6,*) 'Process',rank,'received token',trans, &
     'from process',rank-1
else
  trans=-1
endif
  call MPI_SEND(trans,1,MPI_INT,mod(rank+1,size),0, &
     MPI_COMM_WORLD,ierr)
if(rank .eq. 0) then
  call MPI_RECV(trans,1,MPI_INT,rank-1,0,MPI_COMM_WORLD, &
     status,ierr)
  write(6,*) 'Process',rank,'received token',trans,&
     'from process',size-1
endif
```

# Collective Communication Routines

- MPI_BCAST
- MPI_GATHER
- MPI_SCATTER
- MPI_REDUCE

# Collective communication

- Many algorithms require communication of data between more than pairs of processes.
- In principle doable with `MPI_Send` and `MPI_Recv`
- But much more efficient implementations and convenient interfaces are often possible
  - Special networks for special communication patterns
  - Abstraction of complex communication patterns

# Common features of collective operations

- Blocking (but non-blocking added in version 3.0)
- Involve all processes in the communicator
- Totally ordered within the communicator
- No tags (obviously)
- Designated root process (for many operations)

# Collective communication operations in MPI (2.2)

- `MPI_Barrier`
- `MPI_Bcast`
- `MPI_Gather` (+ variants)
- `MPI_Scatter` (+ variants)
- `MPI_Allgather` (+ variants)
- `MPI_Alltoall` (+ variants)
- `MPI_Reduce`
- `MPI_Allreduce`
- `MPI_Reduce_scatter` (+ variants)
- `MPI_Scan` (+ variants)
- 17 functions in total
- (Version 3.0 doubles that with non-blocking variants)

# MPI_BARRIER

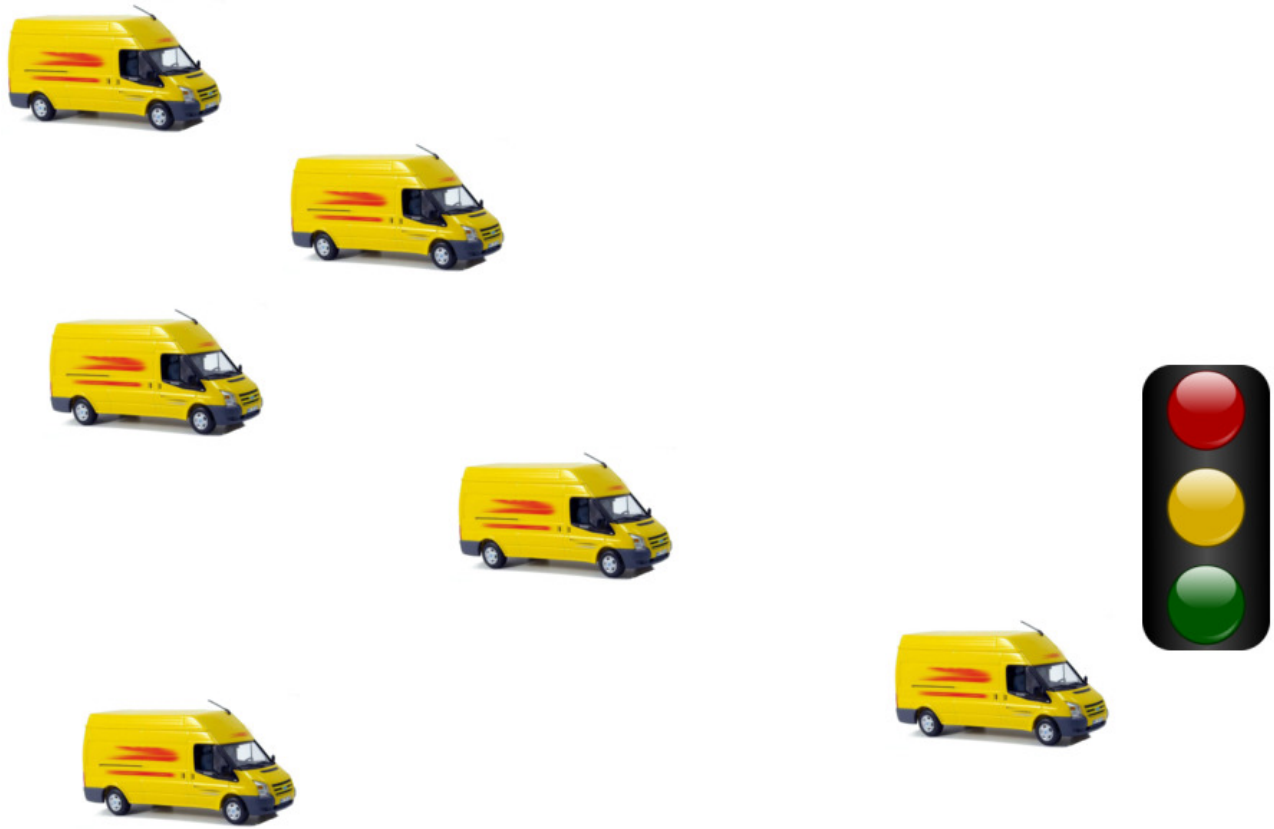

Figure : Barrier operation.

# Barrier

```
MPI_Barrier( MPI_COMM_WORLD );
```

- Synchronizes all processes.
- No-one returns until all have reached the barrier.

# MPI_BCAST

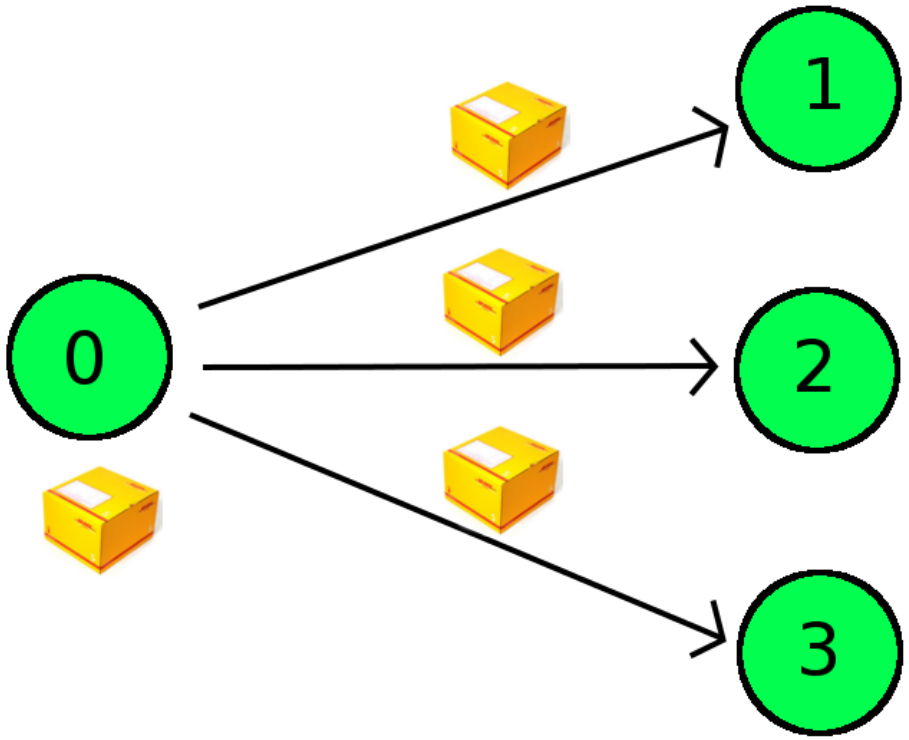


Figure : Broadcast operation.

# Broadcast

```
MPI_Bcast( array, 100, MPI_DOUBLE,
           root, MPI_COMM_WORLD );
```

- Broadcasts (duplicates) to all processes.
- `array` is input on the root
- `array` is output everywhere else

# MPI_GATHER

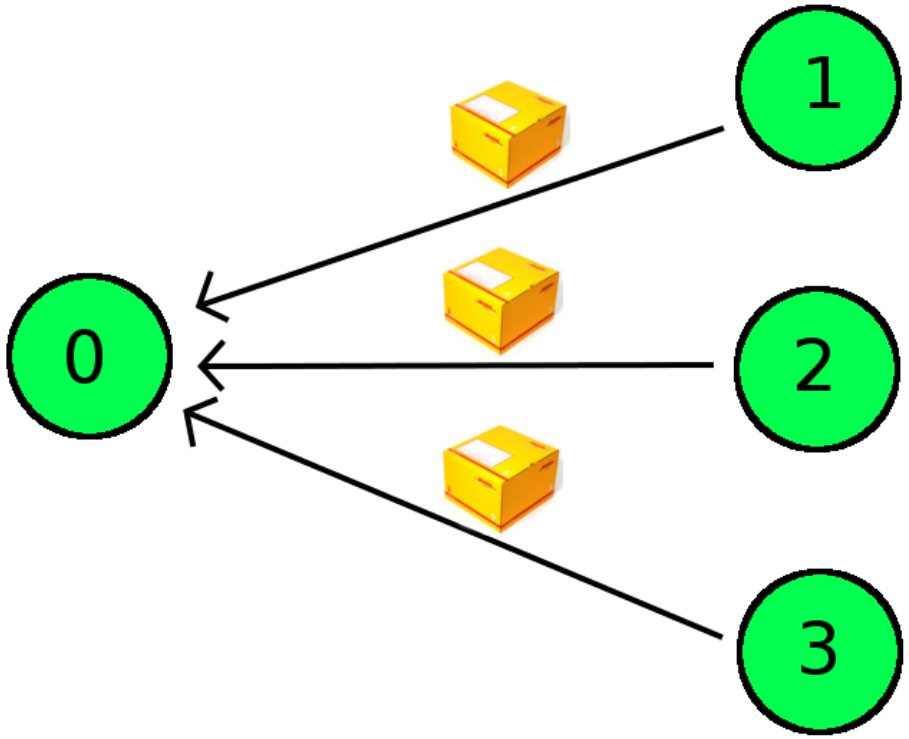


Figure : Gather operation.

# Gather

```
MPI_Gather( send, 100, MPI_INT,
            recv, 100, MPI_INT,
            root, MPI_COMM_WORLD );
```

- Collects on the root a message from each process (including itself)
- `send` is input everywhere
- `recv` is output on the root and everywhere else not referenced
- Related: `MPI_Allgather` returns the result on **all** processes

# Gather: Details

- Q: How are the incoming messages packed in the `recv` array?
- A: In the order of the ranks.

- Q: How large must the `recv` buffer be?
- A: Large enough to acommodate $p$ times the receive count number of elements

# MPI_SCATTER

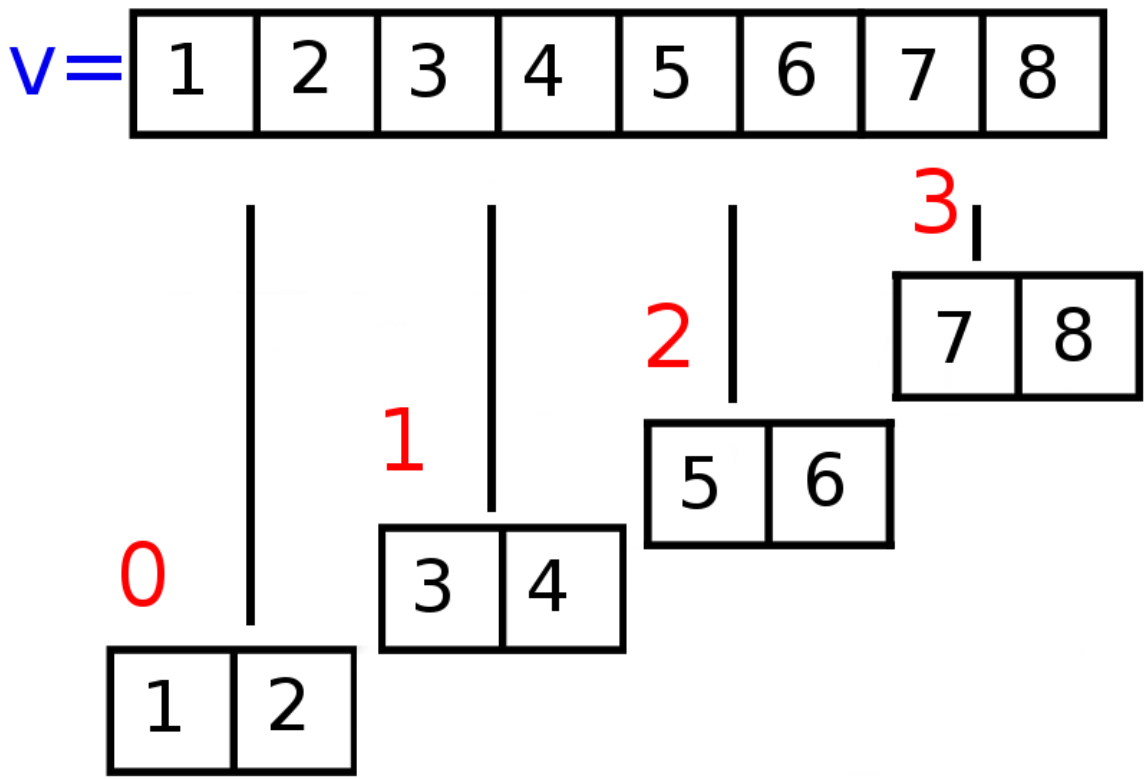


Figure : Scatter operation.

# Scatter

```
MPI_Scatter( send, 100, MPI_INT,
             recv, 100, MPI_INT,
             root, MPI_COMM_WORLD );
```

- The dual of `MPI_Gather`
- Sends from the root one message to each process (including itself)
- `send` is input on the root and everywhere else not referenced
- `recv` is output everywhere

# Scatter: Details

- The root can reuse the send buffer by specifying the constant `MPI_IN_PLACE` instead of the `recv` argument
- (See also `MPI_Gather`)

# Example: Vector dot product

```
call MPI_SCATTER(x,dim2,MPI_REAL,xpart,dim2,MPI_REAL, &
root,MPI_COMM_WORLD,ierr)
call MPI_SCATTER(y,dim2,MPI_REAL,ypart,dim2,MPI_REAL, &
root,MPI_COMM_WORLD,ierr)

 zpart = 0.0
 do i = 1, dim2
 zpart = zpart + xpart(i)*ypart(i)
 enddo
```

# Reduce

```
MPI_Reduce( send, recv, 100, MPI_INT,
            MPI_SUM, root, MPI_COMM_WORLD );
```

- Reduces one message from each process to a single message at the root
- `send` is input everywhere
- `recv` is output on the root
- Related: `MPI_Allreduce` returns the result of **all** processes

# Reduce: Pre-defined reduction operators

- `MPI_MAX`
- `MPI_MIN`
- `MPI_SUM`
- `MPI_PROD` (product)
- `MPI_LAND`, `MPI_LOR`, `MPI_LXOR` (logical and, or, xor)
- `MPI_BAND`, `MPI_BOR`, `MPI_BXOR` (bitwise and, or, xor)
- `MPI_MAXLOC` (max value and location)
- `MPI_MINLOC` (min value and location)

# Reduce: User-defined reduction operators

- Possible to define arbitrary reduction operators
- Must be associative
- Possibly also commutative
- Can operate on arbitrary datatypes (see below)

# Reduce/scatter dot product

```
call MPI_SCATTER(x,dim2,MPI_REAL,xpart,dim2,MPI_REAL, &
root,MPI_COMM_WORLD,ierr)
call MPI_SCATTER(y,dim2,MPI_REAL,ypart,dim2,MPI_REAL, &
root,MPI_COMM_WORLD,ierr)

 zpart = 0.0
 do i = 1, dim2
 zpart = zpart + xpart(i)*ypart(i)
 enddo

call MPI_REDUCE( zpart,z,1,MPI_REAL,MPI_SUM,root, &
MPI_COMM_WORLD,ierr )
 print *, 'Finish processor ', rank
 if( rank == root ) then
 print *, 'Vector product is: ', z
 endif
```

# Scatter columns of a matrix

```
!     Fortran stores this array in column major order, so t
!     scatter will actually scatter columns, not rows.
data sendbuf /1.0, 2.0, 3.0, 4.0, &
              5.0, 6.0, 7.0, 8.0, &
              9.0, 10.0, 11.0, 12.0, &
              13.0, 14.0, 15.0, 16.0 /

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

if (numtasks .eq. SIZE) then
   source = 1
   sendcount = SIZE
   recvcount = SIZE
   call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, recvbuf,
       recvcount, MPI_REAL, source, MPI_COMM_WORLD, ierr)
   print *, 'rank= ',rank,' Results: ',recvbuf
```

*The End!*